01

Testando exceções

Transcrição

Nem sempre queremos que nossos métodos de produção modifiquem estado de algum objeto. Algumas vezes, queremos tratar casos excepcionais. Por exemplo, veja a classe `Avaliador` abaixo. Ela procura pelo maior e menor lance dados em um leilão.

O que aconteceria caso o leilão passado não tenha recebido nenhum lance? O atributo `maiorDeTodos`, por exemplo, ficaria com um número muito pequeno (no caso, `Double.NEGATIVE_INFINITY`). Isso não faz sentido! Nesses casos, muitos desenvolvedores podem optar por lançar uma exceção. Podemos verificar se o leilão possui lances. Caso não possua nenhum lance, podemos lançar uma exceção:

```
public class Avaliador {

    private double maiorDeTodos = Double.NEGATIVE_INFINITY;
    private double menorDeTodos = Double.POSITIVE_INFINITY;
    private List<Lance> maiores;

    public void avalia(Leilao leilao) {
        // lançando a exceção
        if(leilao.getLances().size() == 0) {
            throw new RuntimeException("Não é possível avaliar um leilão sem lances!");
        }

        for(Lance lance : leilao.getLances()) {
            if(lance.getValor() > maiorDeTodos) maiorDeTodos = lance.getValor();
            if (lance.getValor() < menorDeTodos) menorDeTodos = lance.getValor();
        }

        tresMaiores(leilao);
    }

    // código continua aqui...
}
```

Implementado. A pergunta agora é: como testar esse trecho de código? Se escrevermos um teste da maneira que estamos acostumados, o teste falhará, pois o método `avalia()` lançará uma exceção. Além disso, como fazemos o assert? Não existe um `assertException()` ou algo do tipo:

```
@Test
public void naoDeveAvaliarLeiloesSemNenhumLanceDado() {
    Leilao leilao = new CriadorDeLeilao()
        .para("Playstation 3 Novo")
        .constroi();

    leiloeiro.avalia(leilao);

    // como fazer o assert?
}
```

Uma alternativa é fazermos o teste falhar caso a exceção não seja lançada. Podemos fazer isso por meio do método `Assert.fail()`, que falha o teste:

```
@Test
public void naoDeveAvaliarLeiloesSemNenhumLanceDado() {
    try {
        Leilao leilao = new CriadorDeLeilao()
            .para("Playstation 3 Novo")
            .constroi();

        leiloeiro.avalia(leilao);
        Assert.fail();
    }
    catch(RuntimeException e) {
        // deu certo!
    }
}
```

O teste agora passa, afinal o método lançará a exceção, a execução cairá no `catch(RuntimeException e)`, e como não há `assert`s lá dentro, o teste passará. Essa implementação funciona, mas não é a melhor possível.

A partir do JUnit 4, podemos avisar que o teste na verdade passará se uma exceção for lançada. Para isso, basta fazermos uso do atributo `expected`, pertencente à anotação `@Test`. Dessa maneira, eliminamos o try-catch do nosso código de teste, e ele fica ainda mais legível:

```
@Test(expected=RuntimeException.class)
public void naoDeveAvaliarLeiloesSemNenhumLanceDado() {
    Leilao leilao = new CriadorDeLeilao()
        .para("Playstation 3 Novo")
        .constroi();

    leiloeiro.avalia(leilao);
}
```

Veja que passamos a exceção que deve ser lançada pelo teste. Caso a exceção não seja lançada ou não bata com a informada, o teste falhará. Agora já sabemos como testar métodos que lançam exceções em determinados casos.

Melhorando a legibilidade

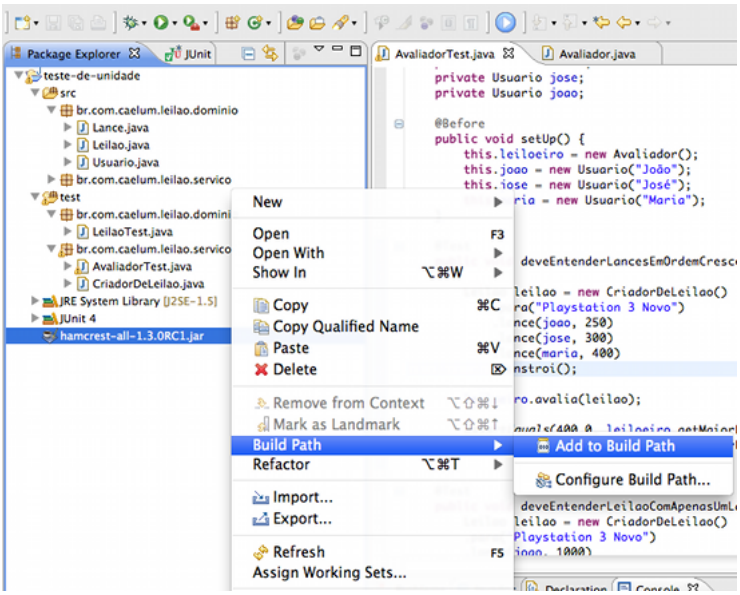
Vamos agora continuar a melhorar nosso código de teste. Nossos testes já estão bem expressivos, mas algumas coisas ainda não são naturais. Uma delas é por exemplo nossos asserts. A ordem exigida pelo JUnit não é "natural", afinal normalmente pensamos no valor que calculamos e depois no valor que esperamos. Além disso, a palavra `assertEquals()` poderia ser ainda mais expressiva. Veja o teste abaixo e compare os dois asserts:

```
class AvaliadorTest {

    @Test
    public void deveEntenderLancesEmOrdemCrescente() {

        Leilao leilao = new CriadorDeLeilao()
            .para("Playstation 3 Novo")
```

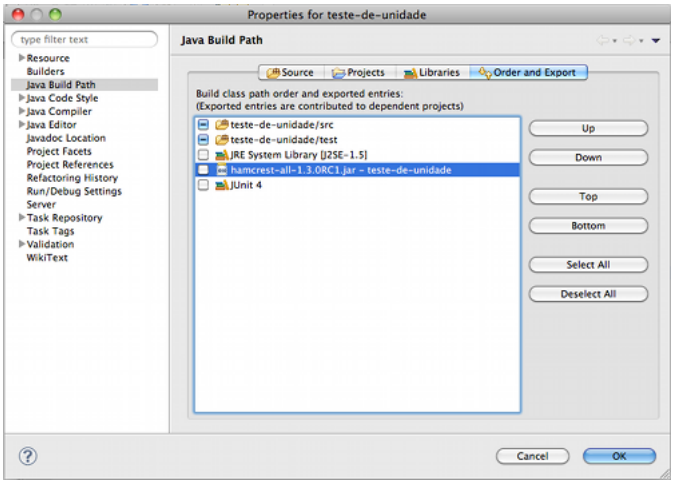
```
                .lance(joao, 250)
                .lance(jose, 300)
                .lance(maria, 400)
                .constroi();

        leiloeiro.avalia(leilao);

        assertThat(leiloeiro.getMenorLance(), equalTo(250.0));
        assertEquals(400.0, leiloeiro.getMaiorLance(), 0.00001);
    }
}
```

Veja que o primeiro assert é muito mais legível. Se lermos essa linha como uma frase em inglês, temos garantia que o menor lance é igual a 250.0. Muito mais legível!

Para conseguirmos escrever asserts como esse, podemos fazer uso do projeto Hamcrest. Ele contém um monte de instruções como essas, que simplesmente nos ajudam a escrever um teste mais claro! Baixe a biblioteca, clicando aqui (http://code.google.com/p/hamcrest/downloads/detail?name=hamcrest-all-1.3.0RC1.jar). Com o .jar em mãos, coloque-o no projeto e adicione-o ao Build Path (clicando com botão direito em cima do .jar, e selecionando Build Path -> Add to Build Path), como demonstrado na figura abaixo:



Além disso, precisamos falar para o Eclipse que a biblioteca do Hamcrest deve ser exportada antes do JUnit. Vamos configurar isso também no JUnit. Clique com o botão direito do mouse sobre o projeto, e selecione Build Path -> Configure Build Path. Em seguida, selecione a aba Order and Export. Coloque o .jar do Hamcrest em cima do JUnit, clicando no botão Up, como mostrado na figura abaixo:



Agora, altere o método de teste na classe `AvaliadorTest` para que o mesmo use as asserções do Hamcrest. Repare nos imports:

```
import static org.junit.Assert.assertEquals;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;
```

Muito mais claro! Vamos continuar. Veja o teste que garante que o avaliador encontra os três maiores lances dados para um leilão:

```
@Test
public void deveEncontrarOsTresMaioresLances() {
    Leilao leilao = new CriadorDeLeilao()
        .para("Playstation 3 Novo")
        .lance(joao, 100)
        .lance(maria, 200)
        .lance(joao, 300)
        .lance(maria, 400)
        .constroi();

    leiloeiro.avalia(leilao);

    List<Lance> maiores = leiloeiro.getTresMaiores();
    assertEquals(3, maiores.size());

    assertEquals(400.0, maiores.get(0).getValor(), 0.00001);
    assertEquals(300.0, maiores.get(1).getValor(), 0.00001);
    assertEquals(200.0, maiores.get(2).getValor(), 0.00001);
}
```

Podemos mudar esse asserts para algo muito mais expressivo:

```
assertThat(maiores, hasItems(
    new Lance(maria, 400),
    new Lance(joao, 300),
    new Lance(maria, 200)
));
```

Veja que estamos conferindo se a lista `maiores` contém os 3 lances esperados! Para que o matcher (o nome na qual esses métodos estáticos `hasItems`, `equalTo`, etc, são chamados) funcione, precisamos implementar o método `equals()` na classe `Lance`:

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Lance other = (Lance) obj;
    if (usuario == null) {
        if (other.usuario != null)
            return false;
    } else if (!usuario.equals(other.usuario))
        return false;
    if (Double.doubleToLongBits(valor) != Double
            .doubleToLongBits(other.valor))
        return false;
    return true;
}
```

O Hamcrest possui muitos outros matchers e você pode conferi-los na documentação do projeto, clicando aqui (http://code.google.com/p/hamcrest/wiki/Tutorial).

Lembre-se sempre de deixar seu teste o mais legível possível! O Hamcrest é uma alternativa!